Fault Tolerance Bounds for Asynchronous Memory Consistency*

Jerry James
jamesj@acm.org

EECS Department
University of Kansas
Lawrence, KS

Raimondas Lencevicius
Raimondas@acm.org

Nokia Research Center
Burlington, MA

Ambuj K. Singh
ambuj@cs.ucsb.edu

CS Department
UC Santa Barbara
Santa Barbara, CA

September 29, 1999

*An earlier version of this work appeared in *WDAG '97*, pp. 200–14 [24].

Proposed running head: Fault Tolerance Bounds, Asynchronous Consistency

Contact author:

Jerry James

EECS Department

University of Kansas

Lawrence, KS 66045

Telephone: (785) 864–7797

Email: jamesj@acm.org

Abstract

A *wait-free* distributed algorithm is one in which every process can complete an operation in a finite number of steps. A generalization of wait-freedom is t-*resilience*, in which every process can complete an operation in a finite number of steps if no more than t processes fail (by stopping). We study the conditions under which algorithms that implement distributed shared memory (DSM) can be implemented resiliently and in a nonblocking manner on *asynchronous* systems, in which there is no known upper bound on message latencies. We derive upper bounds on the number of faults that can be tolerated by an object, based on the consistency of histories of certain forms. From these general bounds, we derive bounds for linearizability, sequential consistency, processor consistency, and some weaker memories. We show that these latter bounds are tight by displaying implementations that achieve them. The proof technique for the upper bounds is of independent interest, due to its applicability to other objects.

Keywords: Distributed shared memory, fault-tolerance bounds, wait-free, t-resilient

1 Introduction

The programming of distributed systems is complex, due to their nondeterministic nature. In the message passing paradigm, the programmer has low-level control, but must explicitly manage shared data. The shared memory paradigm is attractive since it hides the details of such management, making both local and remote data available through read and write operations. Shared memory can be implemented in hardware, software, or a mixture of the two. When a shared memory abstraction is built in software on top of a message passing system, it is called *Distributed Shared Memory* (DSM) [31].

A DSM system provides a contract to the programmer in the form of a *consistency condition* on the shared data. Two well-known consistency conditions are *linearizability* [21, 30] and *sequential consistency* [29], which guarantee that the DSM behaves like a single physical memory module. However, such strong shared memories can have a significant negative impact on the performance of applications, and can limit the scalability of a system [8]. Weaker conditions have been defined that allow optimizations that potentially provide better performance, but can lead to nonsequential behavior. These conditions include processor consistency [2, 18], release consistency [17], pipelined RAM (PRAM) [33], hybrid consistency [7], mixed consistency [1], and causal memory [3].

The capabilities and characteristics of the various memory consistency conditions have been a fruitful topic of study. Both qualitative and quantitative comparisons have been performed. A qualitative comparison differentiates memories by the kinds of problems they can solve, or the properties that any implementation of some consistency condition must have. A quantitative comparison shows the complexity of solving various tasks with different kinds of memory, or the complexity of any implementation of some consistency condition. Some of the known results are:

- For terminating noninteractive programs, the set of possible final states is equivalent for sequential consistency and linearizability, since each requires equivalence with some sequential history.
- Mutual exclusion cannot be solved in a non-cooperative manner for some weak memories [6].

- For data-race-free programs, many weak consistency conditions are equivalent to sequential consistency [1, 3, 17, 41].
- For linearizability and sequential consistency, the sum of the worst-case read and write response times is at least as great as the network delay [8].
- In a partially synchronous model, where message delays are bounded, linearizability incurs higher latency than sequential consistency in the worst case [8, 37, 38].

Although some latency bounds are known for the synchronous and partially asynchronous cases, many real distributed systems are *asynchronous*; there is no known upper bound on message latencies, and neither perfectly reliable failure detectors [12] nor membership services [13] are available. In such systems, slow processes cannot be distinguished from failed processes. Algorithms designed for asynchronous systems have advantages on other kinds of systems as well: they are able to continue processing while fault detection and recovery take place, rather than waiting until such actions have completed. In addition, they are more tolerant of temporary fluctuations in network speeds than algorithms designed to assume (partial) synchrony.

This paper examines the achievable fault tolerances for shared memory implementations on asynchronous message passing systems. The fault tolerance capabilities are measured in terms of t-*resilience*, the ability of every non-failed process to complete an operation in a finite number of steps as long as no more than t processes fail [9, 11, 14, 20, 34]. This is a generalization of *wait-freedom* [19], in which every correct process can complete an operation in a finite number of steps, regardless of the speed with which it or any other process takes steps. Wait-free algorithms provide an extreme form of fault tolerance: a correct process can make progress even if every other process fails. That is, wait-freedom is equivalent to $(N-1)$-resilience in a system of N processes. In this paper, we study t-resilient DSM algorithms.

We analyze the maximum resilience of the read and write operations separately in each case. As we show below, if all operations were required to be equally resilient, linearizability would be indistinguishable from sequential consistency. In some cases, there is a tradeoff—increasing the resilience of one operation requires a decrease in the resilience of the other. This implies that it may be possible in practice to make critical or oft-used operations more resilient at the expense of less critical or less frequently used operations.

We derive upper bounds on the number of process failures that can be tolerated by any implementation of a DSM based on the existence of inconsistent histories of certain forms. In particular, we show the bounds displayed in Table 1. That is, any implementation of linearizability has both read and write operations that tolerate the failure of at most a minority of the processes. Sequential consistency, processor consistency, and some related kinds of memory have at least one operation that tolerates the failure of at most a minority of the processes. Many weak memories have fully wait-free implementations, some of which have been published previously. In all cases, we show that the upper bounds are tight by displaying algorithms that achieve them.

In one sense, the frequent appearance of the quantity $\left\lfloor \frac{N-1}{2} \right\rfloor$ in Table 1 is unsurprising. That is, it is known that, for many problems requiring global coordination, no algorithm can survive the failure of a majority of the processes [5]. However, our proof technique is of independent interest. Rather than argue the bounds for each consistency condition separately in an *ad hoc* manner, we look for patterns. We show that the existence of certain inconsistent histories is sufficient to deduce fault tolerance bounds for a consistency condition. This approach has since been developed further to show that other properties of consistency conditions can be proved in a similar manner.

The rest of the paper is organized as follows. We give our model of distributed shared memory systems in Section 2, along with definitions of asynchrony, t-resilience, and wait-freedom. In Section 3 we derive upper bounds on the fault tolerance of consistency conditions based on the existence of inconsistent histories of a certain form. We also show that these bounds are tight for several consistency conditions by displaying implementations that achieve them. We survey related work in Section 4. We close by remarking on how these results apply to general concurrent objects, and give some notes on future research in Section 5.

Consistency Condition		Snapshot/ Read	Write	Acquire/ Strong Read	Release/ Strong Write
Linearizability	$N \geq 3$	$\left\lfloor \frac{N-1}{2} \right\rfloor$	$\left\lfloor \frac{N-1}{2} \right\rfloor$		
	$N < 3$	$N - 1$	$\left\lfloor \frac{N-1}{2} \right\rfloor$		
Sequential	$M \geq 2$	$\left\lfloor \frac{N-1}{2} \right\rfloor$	$N - 1$		
Consistency		$N - 1$	$\left\lfloor \frac{N-1}{2} \right\rfloor$		
	$M < 2$	$N - 1$	$N - 1$		
Processor	$M \geq 3$	$\left\lfloor \frac{N-1}{2} \right\rfloor$	$N - 1$		
Consistency		$N - 1$	$\left\lfloor \frac{N-1}{2} \right\rfloor$		
	$M < 3$	$N - 1$	$N - 1$		
Release	$M \geq 2$	$N - 1$	$N - 1$	$\left\lfloor \frac{N-1}{2} \right\rfloor$	$N - 1$
Consistency		$N - 1$	$N - 1$	$N - 1$	$\left\lfloor \frac{N-1}{2} \right\rfloor$
(SC)	$M < 2$	$N - 1$	$N - 1$	$N - 1$	$N - 1$
Release	$M \geq 3$	$N - 1$	$N - 1$	$\left\lfloor \frac{N-1}{2} \right\rfloor$	$N - 1$
Consistency		$N - 1$	$N - 1$	$N - 1$	$\left\lfloor \frac{N-1}{2} \right\rfloor$
(PC)	$M < 3$	$N - 1$	$N - 1$	$N - 1$	$N - 1$
Hybrid	$N \geq 3$	$N - 1$	$N - 1$	$\left\lfloor \frac{N-1}{2} \right\rfloor$	$\left\lfloor \frac{N-1}{2} \right\rfloor$
Consistency (LIN)	$N < 3$	$N - 1$	$N - 1$	$N - 1$	$\left\lfloor \frac{N-1}{2} \right\rfloor$
Hybrid	$M \geq 2$	$N - 1$	$N - 1$	$\left\lfloor \frac{N-1}{2} \right\rfloor$	$N - 1$
Consistency		$N - 1$	$N - 1$	$N - 1$	$\left\lfloor \frac{N-1}{2} \right\rfloor$
(SC)	$M < 2$	$N - 1$	$N - 1$	$N - 1$	$N - 1$
Mixed, Causal, etc.		$N - 1$	$N - 1$		

Table 1: Operation Resilience Bounds with N processes & M shared variables

2 Model

In this section, we describe our DSM (Distributed Shared Memory) model, which is similar to those that have appeared elsewhere (e.g., [2, 8, 26, 37]). We define consistency conditions, and also t-resilience and wait-freedom. We also distinguish between synchronous and asynchronous systems, and show that our results apply to more than one kind of asynchronous system.

2.1 DSM Model

In this paper, we investigate the existence of resilient algorithms that implement the shared memory abstraction on distributed systems, or *Distributed Shared Memory* (DSM) [31]. A DSM system consists of a set of n processes, $p_1, p_2, \ldots, p_n$. Each process i executes both an *application program* P_i, and a *memory consistency system* (MCS) M_i. Communication between P_i and M_i is by means of *call* events, in which P_i requests M_i to access the shared memory, and *response* events, in which M_i informs P_i of the outcome of a request. A matching call and response pair form an *operation*. In this paper, we restrict our attention to histories in which each process has at most one pending operation at a time.

Communication between processes, and specifically between the M_i, is by passing messages. In this paper, we consider only systems in which messaging is *reliable*; that is, messages are never lost, misdelivered, delivered multiple times, corrupted, or spontaneously generated by the messaging system. Message channels are not necessarily FIFO. However, we assume that messages are tagged with a sequence number when sent, so that the receiver is able to reconstruct the sending order. Due to the reliability of the messaging system, this means that an MCS can delay processing a message until all earlier messages from the same source have been processed, resulting in effectively FIFO communication. Note that such a strategy has no effect on the fault tolerance of a system, since no extra communication between processes is needed.

The shared memory abstraction presented to the application programs consists of an array of m shared variables, $M[1 \ldots m]$, each of which holds some value from the universe of data values $\mathcal{U}$. Initially, each element of M contains the *initial value* $\perp \in \mathcal{U}$. In this paper, we assume that $\mathcal{U} \subseteq \mathbb{N}$ and $\perp = 0$. All DSM systems described in this paper support the following two operations on M:

$\boldsymbol{Write}\ (\boldsymbol{x} : \boldsymbol{M}, \boldsymbol{v} : \boldsymbol{\mathcal{U}})$ Change the value of shared variable x to value v.

$\boldsymbol{Read}\ (\boldsymbol{x} : \boldsymbol{M}) \rightarrow \boldsymbol{\mathcal{U}}$ Return the value of shared variable x.

In addition, we consider a *snapshot* operation, which is equivalent to simultaneous read operations on all shared memory locations. This operation will be useful for providing succinct implementations later in the paper. Perhaps surprisingly, the addition of this operation does not change the lower bound for any consistency condition, and the resilience bounds for this operation are always the same as those for the read operation. These results are shown later. The snapshot operation is defined as follows:

$\boldsymbol{Snapshot}\ () \rightarrow \boldsymbol{\mathcal{U}}\,[\boldsymbol{m}]$ Return the value of all shared variables in one atomic action. That is, this operation is equivalent to m simultaneous reads.

Some shared memory systems also support synchronization operations (e.g., *acquire* and *release* for release consistent systems), or *strong* and *weak* variations on the operations (e.g., slow memory, hybrid consistency).

Each call event and message arrival triggers a deterministic algorithm at the receiving MCS that determines both changes in local state, and the messages sent to other processes. A *history* H of a DSM system is a record of the events that took place during some execution (i.e., the local state changes, and the sending and receiving of messages), together with the real time of each event. A *local history* H_i of process i consists exactly of the events of process i from some history H. A *history restricted to variable* x consists exactly of the events of all processes corresponding to operations on shared variable x, written $H|_x$.

In this paper, we typically give *reduced* shared memory histories, with local state changes omitted. Since this brings operation invocation and response event pairs into proximity, we merge them and speak of *operations*. In this case, we use the following shorthand. A completed write of v to shared variable x is written $W(x)v$. A completed read of variable x returning value v is written $R(x)v$. A completed snapshot of the shared memory returning array M is written $S()M$.

2.2 Consistency Conditions

For a reduced shared memory history H, let $ops\,(H)$ represent the set of operations invoked in H. Let $op_1 \longrightarrow op_2$ indicate that the response event of operation op_1 occurred before the call event of operation op_2. The relation $\longrightarrow$ is easily shown to be an irreflexive partial order on $ops\,(H)$. Since each process has at most one incomplete operation at a time, the operations invoked by process i are totally ordered by $\longrightarrow$. Let $\xrightarrow{i}$ be the local (total) order on the operations at process i. A *linearization* of H is a total order $\xrightarrow{L}$ on $ops\,(H)$ such that the sequential read/write semantics are preserved, and $op_1 \longrightarrow op_2 \Rightarrow op_1 \xrightarrow{L} op_2$. A *serialization* of H is a total order $\xrightarrow{S}$ on $ops\,(H)$ such that the sequential read/write semantics are preserved, and $\left\langle \exists i :: op_1 \xrightarrow{i} op_2 \right\rangle \Rightarrow op_1 \xrightarrow{S} op_2$. Note that any linearization of H is also a serialization of H.

A shared memory system must be able to respond to any pattern of operation invocations. It has freedom only to choose return values in some consistent way. We capture this idea with the following definitions.

Definition 1. A *shared memory history skeleton* is a shared memory history with all operation return values removed. The *completion* of a history skeleton is an assignment of one return value to each operation; a *complete* history is one with all return values specified.

Definition 2. A *consistency condition* C is a set of reduced shared memory histories such that, for every shared memory history skeleton S, C contains some completion of S. A history H is said to be C*-consistent* if $H \in C$; otherwise it is C*-inconsistent*.

That is, *every* pattern of operation invocations must have some set of return values that makes it consistent.

Consistency conditions are generally defined in terms of linearizations and serializations of some subset of the operations in a history. We will see several examples of consistency conditions later in the paper. We now give an important characteristic of consistency conditions.

Definition 3. Let C be a shared memory consistency condition. If the set of histories in C is closed under the renaming of processes, then C is *oblivious*.

Informally, an oblivious consistency condition is one that depends only on the operations that are invoked, not on the processes that did the invoking. All commonly studied consistency conditions, including all the conditions in this paper, are oblivious. This is similar to the notion of obliviousness that arises in the study of wait-free algorithms (e.g., [11, 40]). There, each individual operation can be invoked by any process, in contrast to our definition, which allows only the renaming of a process; i.e., changing p_i to p_k affects all operations in H_i (which are then in H_k), rather than changing the invoking process for individual operations. Note that, in general, nonlinearizable shared memories do not meet the stronger obliviousness condition. The weaker condition stated above is sufficient for our purposes.

2.3 *t*-resilience and Wait-freedom

A central concept in this paper is that of t-resilience, the ability to withstand up to t process failures. We are concerned with stop failures, in which processes simply cease taking any action. In any history, processes that stop are referred to as *faulty*; the other processes are *correct*. A *t-faulty history* is one in which there are exactly t faulty processes. We extend our notion of a complete history in this case. A history H is *complete* if all incomplete operations occur on faulty processes.

Definition 4. Let D be a DSM system. We say that operation op of D is *t-resilient* if every complete s-faulty history H of D, $s \leq t$, satisfies the following:

- Every invocation of operation op on a correct process has a matching response (we say that op is never *blocked*); and
- $H \in C$ (i.e., H is consistent).

An algorithm that is $(N-1)$-resilient in a system of N processes is *wait-free* [19] in that system. A DSM system in which every operation is t-resilient (resp. wait-free) is said to be t-resilient (resp. wait-free).

2.4 Asynchrony

Earlier work has distinguished between *synchronous*, *partially asynchronous*, and *asynchronous* systems, although the use of the terms has not been uniform. In contrast to synchronous systems, in which each message has a fixed latency, and asynchronous systems, in which there is no maximum latency, *partially asynchronous* systems exhibit a range of message latencies. The usual definition is that each message has a maximum latency d and an *uncertainty factor* u; i.e., every message takes time between $d - u$ and d to arrive.

We are interested in studying DSM implementations in asynchronous systems, in which there is no maximum latency on messages. For such systems to be interesting, we add the stipulation that all messages are eventually delivered. That is, given any finite prefix of a history of an asynchronous system, there is some finite value m that represents the *maximum observed message latency* for that prefix. This value is not bounded above, but is always finite.

We now show that we can, in fact, study a simpler model. For certains types of systems (including DSM systems, as we have defined them), a partially asynchronous system with an unknown maximum latency d is equivalent to an asynchronous system. Not only does this imply that algorithms that are correct for the latter are correct for the former, it also implies that algorithms that are *incorrect* for the former are incorrect for the latter. This becomes important for our impossibility proofs, where we show that no algorithm exists for asynchronous systems in some circumstances. The equivalence then implies that no algorithm exists for partially asynchronous systems with an unknown upper bound.

There is a fine line separating asynchronous systems from partially asynchronous systems with unknown d. The difference can be seen only in infinite histories of asynchronous systems where message latencies continually grow. Any finite prefix of such a history has a finite bound, but there is no limit point for those latencies; hence the history is not valid for any partially asynchronous system. For distributed shared memories, however, this distinction is immaterial, since correctness depends only on the relative order of events, and not on the absolute times of those events.

The idea is to show that any history of an asynchronous system D can be *scaled* to a history of a partially asynchronous system D'. The scaling process preserves all orderings between message sends and receives, but yields a known upper bound on message latencies. Since orderings are preserved, the two histories are indistinguishable for systems whose correctness depends only on the relative ordering of events, rather than the absolute times of those events. This is a simple matter for finite histories, where the maximum observed latency plays the role of maximum latency. However, the result follows even for infinite histories, which is necessary for our technique to apply to liveness properties.

Consider a (possibly infinite) asynchronous history H. For each event $e \in H$, let $d(e)$ be the longest latency of any message sent before or simultaneously with e (including e itself, if e is a message send event). Note that d is monotonically nondecreasing with time. Then d induces a partition P_H of H such that, for each $p \in P_H$ and $e \in p$, $f \in p$ iff $d(e) = d(f)$. We write $d(p) = t$ to indicate that, for all $e \in p$, $d(e) = t$. Since d is monotonically nondecreasing, the partition is totally ordered with respect to time. Hence, we write $p_1, p_2, \ldots$ and say that $p_i < p_j$ iff $i < j$ (implying that $d(p_i) < d(p_j)$). Finally, let $\mathit{begin}(p)$ be the earliest time of any event in p, or zero for the first partition (p_1). For each $p_i \in P_H$ and constant $u > 0$, define the following quantities:

$$
\begin{aligned}
\mathit{length}\,(p_i) &= \begin{cases} \mathit{begin}\,(p_{i+1}) - \mathit{begin}\,(p_i) & \text{if } p_{i+1} \text{ exists} \\ \infty & \text{otherwise} \end{cases} \\
\mathit{scale}\,(p_i, u) &= \min\left(1, \frac{u}{d\,(p_i)}\right) \qquad \{\text{Note that } \mathit{scale}\,(p_i, u) \in (0, 1]\} \\
\mathit{shift}\,(p_i, u) &= \mathit{begin}\,(p_i) \cdot (1 - \mathit{scale}\,(p_i, u)) - \sum_{j=1}^{i-1} \mathit{length}\,(p_j) \cdot (1 - \mathit{scale}\,(p_j, u))
\end{aligned}
$$

First we give a technical lemma that reduces the complexity of the following proofs.

Lemma 5. *If $i < j$, then $\mathit{shift}\,(p_j, u) - \mathit{shift}\,(p_i, u) = \sum_{k=i}^{j-1} \mathit{begin}\,(p_{k+1}) \cdot (\mathit{scale}\,(p_k, u) - \mathit{scale}\,(p_{k+1}, u))$*

Proof. By simple arithmetic manipulation, we get the following:

$$\mathit{shift}\,(p_j, u) - \mathit{shift}\,(p_i, u)$$

$$
\begin{aligned}
&= \mathit{begin}\,(p_j)\cdot(1-\mathit{scale}\,(p_j,u)) - \mathit{begin}\,(p_i)\cdot(1-\mathit{scale}\,(p_i,u))\\
&\quad -\sum_{k=1}^{j-1}\mathit{length}\,(p_k)\cdot(1-\mathit{scale}\,(p_k,u)) + \sum_{l=1}^{i-1}\mathit{length}\,(p_l)\cdot(1-\mathit{scale}\,(p_l,u))\\
&= \mathit{begin}\,(p_j)\cdot(1-\mathit{scale}\,(p_j,u)) - \mathit{begin}\,(p_i)\cdot(1-\mathit{scale}\,(p_i,u))\\
&\quad -\sum_{k=i}^{j-1}(\mathit{begin}\,(p_{k+1})-\mathit{begin}\,(p_k))\cdot(1-\mathit{scale}\,(p_k,u))\\
&= \mathit{begin}\,(p_j)\cdot(1-\mathit{scale}\,(p_j,u)) - \mathit{begin}\,(p_i)\cdot(1-\mathit{scale}\,(p_i,u))\\
&\quad +\sum_{k=i}^{j-1}(\mathit{begin}\,(p_{k+1})\cdot\mathit{scale}\,(p_k,u)+\mathit{begin}\,(p_k)\cdot(1-\mathit{scale}\,(p_k,u))-\mathit{begin}\,(p_{k+1}))\\
&= \sum_{k=i}^{j-1}\mathit{begin}\,(p_{k+1})\cdot\mathit{scale}\,(p_k,u)+\sum_{l=i+1}^{j}\mathit{begin}\,(p_l)\cdot(1-\mathit{scale}\,(p_l,u))-\sum_{m=i}^{j-1}\mathit{begin}\,(p_{m+1})\\
&= \sum_{k=i}^{j-1}\mathit{begin}\,(p_{k+1})\cdot\mathit{scale}\,(p_k,u)-\sum_{l=i+1}^{j}\mathit{begin}\,(p_l)\cdot\mathit{scale}\,(p_l,u)\\
&= \sum_{k=i}^{j-1}\mathit{begin}\,(p_{k+1})\cdot(\mathit{scale}\,(p_k,u)-\mathit{scale}\,(p_{k+1},u))
\end{aligned}
$$

□

Lemma 6. $i \le j \Rightarrow \mathit{scale}\,(p_i,u) \ge \mathit{scale}\,(p_j,u) \wedge \mathit{shift}\,(p_i,u) \le \mathit{shift}\,(p_j,u)$

Proof. The result for *scale* follows immediately from the fact that $d(p_i)$ is monotonically nondecreasing in i. For *shift*, the result follows from Lemma 5, the result for *scale*, and the nonnegativity of $\mathit{begin}(p)$. □

We now construct an asynchronous history $H' = \mathit{compress}\,(H,u)$ with maximum message latency bounded above by some constant u and preserving the event orderings of H. For each event $e \in H$ such that e is in partition p, let $e' \in H'$ be e, except $\mathit{time}\,(e') = \mathit{time}\,(e)\cdot\mathit{scale}\,(p,u)+\mathit{shift}\,(p,u)$.

Lemma 7. *Let $H' = \mathit{compress}\,(H,u)$. Then $\mathit{time}(e) < \mathit{time}(f)$ iff $\mathit{time}(e') < \mathit{time}(f')$.*

Proof. If e and f are in the same partition, then they are scaled by the same positive amounts and shifted by the same nonnegative amounts, thereby preserving their ordering. We now show that, for adjacent partitions p_i and p_{i+1}, all scaled events in p_i precede any scaled event of p_{i+1}, thereby preserving event orderings between partitions. Since event $\mathit{begin}\,(p_{i+1})$ follows all events of p_i in time, it is sufficient to show the following:

$$
\begin{aligned}
&\mathit{scale}\,(p_i,u)\cdot \mathit{begin}\,(p_{i+1})+\mathit{shift}\,(p_i,u)\\
&\quad= \mathit{scale}\,(p_i,u)\cdot \mathit{begin}\,(p_{i+1})+\mathit{shift}\,(p_{i+1},u)+\mathit{begin}\,(p_i)\cdot(1-\mathit{scale}\,(p_i,u))\\
&\qquad -\mathit{begin}\,(p_{i+1})\cdot(1-\mathit{scale}\,(p_{i+1},u))+\mathit{length}\,(p_i)\cdot(1-\mathit{scale}\,(p_i,u))\\
&\quad= \mathit{scale}\,(p_i,u)\cdot \mathit{begin}\,(p_{i+1})+\mathit{shift}\,(p_{i+1},u)+\mathit{begin}\,(p_i)-\mathit{scale}\,(p_i,u)\cdot\mathit{begin}\,(p_i)\\
&\qquad -\mathit{begin}\,(p_{i+1})+\mathit{scale}\,(p_{i+1},u)\cdot\mathit{begin}\,(p_{i+1})+(\mathit{begin}\,(p_{i+1})-\mathit{begin}\,(p_i))\cdot(1-\mathit{scale}\,(p_i,u))\\
&\quad= \mathit{scale}\,(p_{i+1},u)\cdot\mathit{begin}\,(p_{i+1})+\mathit{shift}\,(p_{i+1},u)
\end{aligned}
$$

□

Lemma 8. *The latency of every message in $compress(H,u)$ is bounded above by u.*

Proof. Let e be the sending event for a message, and f the receiving event. Suppose that $e\in p_i$ and $f\in p_j$, $i\le j$. Then $time(f)-time(e)\le d(p_i)$. The latency of the message in H' is given by:

$$
\begin{aligned}
&\mathit{time}\,(f')-\mathit{time}\,(e')\\
&\quad= \mathit{scale}\,(p_j,u)\cdot\mathit{time}\,(f)+\mathit{shift}\,(p_j,u)-\mathit{scale}\,(p_i,u)\cdot\mathit{time}\,(e)-\mathit{shift}\,(p_i,u)\\
&\quad= \mathit{scale}\,(p_j,u)\cdot\mathit{time}\,(f)-\mathit{scale}\,(p_i,u)\cdot\mathit{time}\,(e)+\sum_{k=i}^{j-1}\mathit{begin}\,(p_{k+1})\cdot(\mathit{scale}\,(p_k,u)-\mathit{scale}\,(p_{k+1},u))\\
&\quad\le \mathit{scale}\,(p_j,u)\cdot\mathit{time}\,(f)-\mathit{scale}\,(p_i,u)\cdot\mathit{time}\,(e)+\sum_{k=i}^{j-1}\mathit{begin}\,(p_{k+1})\cdot(\mathit{scale}\,(p_i,u)-\mathit{scale}\,(p_j,u))\\
&\quad= \left(\mathit{time}\,(f)-(j-i)\cdot\sum_{k=i}^{j-1}\mathit{begin}\,(p_{k+1})\right)\cdot\mathit{scale}\,(p_j,u)\\
&\qquad -\left(\mathit{time}\,(e)-(j-i)\cdot\sum_{k=i}^{j-1}\mathit{begin}\,(p_{k+1})\right)\cdot\mathit{scale}\,(p_i,u)\\
&\quad\le \left(\mathit{time}\,(f)-(j-i)\cdot\sum_{k=i}^{j-1}\mathit{begin}\,(p_{k+1})\right)\cdot\mathit{scale}\,(p_i,u)\\
&\qquad -\left(\mathit{time}\,(e)-(j-i)\cdot\sum_{k=i}^{j-1}\mathit{begin}\,(p_{k+1})\right)\cdot\mathit{scale}\,(p_i,u)
\end{aligned}
$$

$$
\begin{aligned}
&= (time(f) - time(e)) \cdot scale(p_i, u) \\
&\leq d(p_i) \cdot \frac{u}{d(p_i)} \\
&= u
\end{aligned}
$$

□

Theorem 9. *An algorithm whose correctness depends only on the relative ordering of events, and no the absolute times of those events, is correct for asynchronous systems iff it is correct for partially asynchronous systems with unknown d.*

Proof.

(Partially asynchronous $\Longrightarrow$ asynchronous)

Suppose not. Then there is some incorrect history H of the implementation on an asynchronous system. Let $H' = compress(H, u)$ for some positive number u. Then by Lemma 7, the events of H and H' occur in the same order with respect to each other, and by Lemma 8, all message delays are bounded above by u. This implies that H' is an incorrect history of the implementation on a partially asynchronous system, a contradiction.

(Asynchronous $\Longrightarrow$ partially asynchronous)

Consider a history H of the algorithm on some partially asynchronous system with (unknown) upper bound d on message latencies. H is also a possible history of the algorithm on an asynchronous system with maximum observed message latency of at most d for any finite prefix.

□

3 Fault Tolerance Bounds

In this section, we analyze the fault tolerance bounds for several shared memory consistency conditions. We give general conditions that imply upper bounds, then show the application of those bounds to specific consistency conditions. We show that all such bounds are tight by displaying implementations that achieve them. We begin with conditions on wait-free implementations.

3.1 Wait-free Operations

The upper bounds derived in the next section are based on the existence of histories in which processes issue streams of operations without coordinating with each other. We call such streams of operations *solo executions*; informally, they are sequences of operations that are not affected by concurrent non-local operations.

Definition 10. Let H be a complete history on one process (see Definition 1). Then H is called a *solo execution*.

Definition 11. The *merge* of solo executions $H_1, \ldots, H_k$ is a history $H' = \sqcup_{i=1}^{N} H_i$ such that each local history $H'_i = H_i$.

We refer to a history H as a *merged history* if it is equal to the merge of some set of solo executions.

Theorem 12. *Let C be a consistency condition. If the merge of any set of C-consistent solo executions is C-consistent, then C has a wait-free implementation.*

Proof. Consider the trivial implementation I_{Triv} that never communicates between processes; that is, each process operates in isolation. Any history of I_{Triv} is equal to the merge of some set of C-consistent solo executions, since 1-process histories are always consistent. By supposition, then, any history of I_{Triv} is C-consistent. Hence, I_{Triv}, a wait-free algorithm, is a correct implementation of C. $\square$

The converse does not hold, since the merge of infinite histories may violate liveness conditions. However, if we add a finiteness restriction, the proof follows easily.

Theorem 13. *If consistency condition C has a wait-free implementation, then the merge of any set of finite C-consistent solo executions is C-consistent.*

Proof. Let I_C be a wait-free implementation of C, and let H be the merge of some set of finite C-consistent histories. Since I_C is wait-free, each process must complete any incomplete operation within a finite amount of time, whether messages are received from other processes or not. By the total asynchrony of the system, delay all message receptions until after all operations in H have completed. Thus, H is a possible history of I_C, so H is C-consistent by the correctness of I_C. □

3.2 Upper Bound Conditions

In this section, we give conditions on consistent histories that imply upper bounds on the number of faults that can be tolerated by any implementation of a DSM system. In fact, we give conditions that imply upper bounds for individual operations. In the next section, we show that these bounds are tight for many commonly studied consistency conditions, by displaying implementations that achieve them.

3.2.1 Inconsistent Merges

We now present the first of two conditions which imply a nontrivial upper bound on the fault tolerance of an operation. The first condition says that if all merged histories are not consistent, then some operation has a nontrivial upper bound.

Lemma 14. *Let $op_1, \ldots, op_m$ be t_1-, ..., t_m-resilient operations, respectively. Then a process can execute to completion any sequence of operations from $op_1, \ldots, op_m$ without receiving any message from $\min_{i=1}^{N}\{t_i\}$ other processes.*

Proof. Let $t = \min_{i=1}^{N}\{t_i\}$. Suppose that t processes fail. Then, since each individual operation op_i is at least t-resilient, all operations are able to complete successfully, even if t of the processes fail. In particular, all operations can complete if no messages are received from t processes. □

Theorem 15. *Let C be an oblivious consistency condition and $\mathcal{O}$ a set of operations. If there exists a finite realizable C-inconsistent merged history H on k processes such that every operation in $\mathcal{O}$ is invoked at least once, then some operation in $\mathcal{O}$ is at most $\left\lfloor \frac{(k-1)N-1}{k} \right\rfloor$-resilient in any implementation of C.*

Proof. The proof is by contradiction. Suppose that there exists an implementation I_C of C such that all operations in $\mathcal{O}$ are at least $\left\lceil \frac{(k-1)N}{k} \right\rceil$-resilient. Let G_1, G_2, $\ldots$, G_k be disjoint sets of processes such that $|G_i| = \left\lfloor \frac{N}{k} \right\rfloor$ (since $k \cdot \left\lfloor \frac{N}{k} \right\rfloor \leq N$). For each G_i, choose some process $P_i \in G_i$. By Lemma 14 and the obliviousness of C, each P_i can execute one solo execution H_i from H while communicating with only the processes in G_i, since $N - \left\lceil \frac{(k-1)N}{k} \right\rceil = \left\lfloor \frac{N}{k} \right\rfloor$. Messages from P_i to processes outside of G_i are delayed until after the completion of H. Thus, by Lemma 14, H is a possible history of I_C, contrary to the assumption that I_C implements C. □

3.3 Inconsistent Retimings

We now give the second condition that implies a nontrivial upper bound for some operation. The basic idea is that highly resilient concurrent operations can be arbitrarily ordered in real time, so all such orderings must be consistent. Surprisingly, this holds even if a less resilient operation executes concurrently with the highly resilient operations. This is due to the fact, which we now demonstrate, that operations never wait for less resilient operations (in contrast to wait-freedom, in which no operation waits for any other).

Lemma 16. *Let D be a DSM system with a t-resilient operation o_1 and an s-resilient operation o_2, $t > s$. Then any execution of o_1 can complete in a finite number of steps, regardless of concurrent executions of o_2, if no more than t processes fail.*

Proof. The proof is by contradiction. Suppose that executions of o_1 can be blocked by concurrent executions of o_2. Then o_1 is not wait-free, so $t < N - 1$. Consider a history in which process 1 invokes o_1, process 2 simultaneously invokes o_2, and $s + 1 \leq t < N - 1$ other processes fail before either operation completes. Since o_2 is only s-resilient, its progress is blocked. Since o_2 can block o_1, the progress of o_1 is also blocked, even though only $s + 1 \leq t$ processes failed, a contradiction to the t-resilience of o_1. □

Due to the ability of highly resilient operations to run unimpeded by less resilient operations, the system cannot enforce orderings between concurrent executions of the highly resilient operations. For example, in some cases solo executions are consistent, as long as they involve a restricted set of operations (e.g., concurrent read operations can execute in isolation in a DSM, since the state of the DSM is unchanged.) However, if the isolation "ends" at some point, that is, if some remote state change becomes visible, then it may be necessary for all processes to end their isolation together. For example, in a linearizable DSM, once a write operation becomes visible (its value is returned by some read operation), then all subsequent read operations by any process must reflect that write operation (or a later write to the same location). In such cases, the read operations must respect a time ordering, and so the read operation has a nontrivial upper bound on its resilience.

We now give the notion of a retiming, in which the cross-process orderings of some operations are changed.

Definition 17. Let H be a history on N processes. Let $o_1, \ldots, o_k$ be operations on k distinct processes in H. History H' is a *retiming* of H if:

- H is identical to H' except that the times of operations $o_1, \ldots, o_k$ are different; and
- each process invokes operations in the same order in H and H' ($H_i[1\ldots] = H'_i[1\ldots]$).

The operations $o_1, \ldots, o_k$ are the *retimed operations* of H'. The operation names represented in $o_1, \ldots, o_k$ are the *types of retimed operations*.

Theorem 18. *Let C be an oblivious consistency condition. If there exists a C-consistent history H and a C-inconsistent history H' such that:*

- *H' is a retiming of H with k retimed operations; and*
- *each retimed operation is the first operation invoked by the respective process;*

then at least one of the types of retimed operations is at most $\left\lfloor \frac{(k-1)N-1}{k} \right\rfloor$-resilient in any implementation of C.

Proof. The proof is by contradiction. Suppose that there exists an implementation I_C of C such that all types of retimed operations are at least $\left\lceil \frac{(k-1)N}{k} \right\rceil$-resilient. Let $G_1, G_2, \ldots, G_k$ be disjoint sets of processes such that $|G_i| = \left\lfloor \frac{N}{k} \right\rfloor$. For each G_i, choose some process $P_i \in G_i$. By Lemma 14 and the obliviousness of C, each P_i can execute one of the retimed operations from H while communicating with only the processes in G_i. This is so even if a less resilient operation is running concurrently, by Lemma 16. Messages from P_i to processes outside of G_i are delayed until the beginning of P_i's next operation, or the end of the history if P_i does not invoke any more operations. Then any retiming of this history, pursuant to the restrictions above, results in the same message receptions at all processes. Thus, both H and H' are possible histories of I_C, contrary to the assumption that I_C implements C. □

3.4 Bounds for DSMs

In this section, we show that the upper bounds derived in the previous section are tight for several consistency conditions that appear in the literature. We do so by showing which upper bounds apply, then displaying implementations that achieve those bounds.

3.4.1 Linearizability

A DSM implementation guarantees linearizability [21, 30] if a linearization exists for every history of the system. Linearizability is a *local property*; if each individual object in a system is linearizable, then the entire system is linearizable. We use this property below when we show how to construct a linearizable register. A collection of such registers forms a linearizable shared memory.

It is known that no linearizable register can be implemented to tolerate more than $\left\lfloor \frac{N-1}{2} \right\rfloor$ failures (see, for example, [35, p. 581]). In this section, we show that that failure bound holds individually for both the read and the write operation. We also show that this bound is tight by exhibiting an implementation that achieves it. We analyze the read, write, and snapshot operations separately.

Theorem 19. *In any implementation of linearizability, the write operation is at most* $\left\lfloor \frac{N-1}{2} \right\rfloor$*-resilient.*

Proof. Consider the histories of Figure 1. They are retimings of one another, with 2 retimed operations (the writes). The history of Figure 1(a) is linearizable; the required linearization is the order of operation invocation. However, the history of Figure 1(b) is not linearizable. Since the write operations are separated in time, the read must return 1 to be consistent. Since the write operation is the only retimed operation type, the result follows from Figure 18. □

Theorem 20. *In any implementation of linearizability on at least 3 processes, the read and snapshot operations are at most* $\left\lfloor \frac{N-1}{2} \right\rfloor$*-resilient.*

Proof. Consider the histories of Figure 2. They are retimings of one another, with 2 retimed operations. The history of Figure 2(a) is linearizable; the required linearization is the earlier read operation, the write operation, and then the later read operation. However, the history of Figure 2(b) is not linearizable. Since the read operations are separated in time, the second read must return 1 to be consistent. Since the read operation is the only retimed operation type, and the retimed operations are on 2 of 3 processes, the result for the read operation follows from Theorem 18.

Since the snapshot operation implements the read operation in a wait-free manner (take a snapshot, then return the single value of interest), the snapshot operation cannot be any more resilient than the read operation. Hence, it has the same upper bound. □

Theorem 21. *There exists an implementation of linearizability with* $\left\lfloor \frac{N-1}{2} \right\rfloor$*-resilient write, read, and snapshot operations.*

Proof. A construction is given in [4] for a 1-writer multi-reader atomic register in an asynchronous system. This register has the property that both the read and the write operation are $\left\lfloor \frac{N-1}{2} \right\rfloor$-resilient. In [32], a construction is given of a multi-writer multi-reader atomic register from 1-writer multi-reader atomic registers[1]. This construction has the property that it is wait-free with respect to the underlying 1-writer register construction. That is, implementation of both the read and write operations consists of a series of operations on 1-writer registers, with no explicit waits. By Lemma 14, composing the above two protocols yields an

[1]The main construction is from 1-writer 1-reader registers to multi-writer multi-reader registers; the construction we use is a projection onto the simpler problem of implementing from 1-writer multi-reader registers.

implementation of a multi-writer multi-reader atomic register in an asynchronous distributed system with $\left\lfloor \frac{N-1}{2} \right\rfloor$-resilient read and write operations.

Since linearizability is a local property, a collection of these registers constitutes a linearizable shared memory with the desired fault tolerance bounds for the read and write operations. A snapshot operation can be implemented on such a collection of registers in a wait-free manner from the supplied read and write operations [22, 23]. Hence, the snapshot operation has the same resilience behavior as the read operation, yielding the desired result. $\square$

Both unbounded and bounded implementations are given in [4, 32]. In particular, the execution of either a read or write operation in the bounded construction of [4] requires $O(N)$ messages, each of size $O(N^3)$, and $O(1)$ messaging rounds; $O(N^4)$ local memory is required on each reader, and $O(N^5)$ local memory is required on the writer. The bounded construction of [32] requires $O(N)$ 1-writer registers per multi-writer register; each process is writer for 1 register and reader for $O(N)$. Each process also has $O(N)$ control bits. Each 1-writer register is accessed $O(1)$ times per operation. A naive composition of the two protocols thus yields an implementation requiring $O(N^5)$ local memory on each process; each read or write operation results in the sending of $O(N^2)$ messages, each of size $O(N^3)$, in $O(N)$ messaging rounds.

Note that Theorem 20 does not hold for implementations with only 2 processes. In that case, although the write operation must be 0-resilient by Theorem 19, the read operation can be wait-free, thereby bettering the bound of Theorem 21. The idea is that the 0-resilient write operation does not end until both processes have the new value. Hence, read operations can be purely local (wait-free). We omit the details, as the construction is relatively uninteresting.

3.4.2 Sequential Consistency

A DSM implementation guarantees sequential consistency [29] if a serialization exists for every history of the system. Sequential consistency is strictly weaker than linearizability. It only requires that every history be equivalent to *some* sequential history. Unlike linearizability, it does not require that global time orderings between operations be observed.

Theorem 22. *In any implementation of sequential consistency with at least two shared variables, either the read operation or the write operation is at most* $\left\lfloor \frac{N-1}{2} \right\rfloor$*-resilient.*

Proof. Consider the history of Figure 3, the merge of two solo executions. It is not sequentially consistent; otherwise at least one read operation would return 1. The result then follows from Theorem 15. □

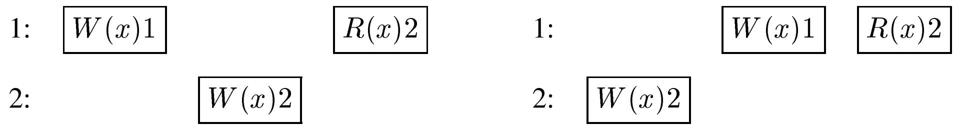
Theorem 22 does not hold for implementations of sequential consistency with only one shared variable. In that case, sequential consistency collapses to the weaker condition of cache consistency, or coherence [18]. As we show in Section 3.4.7, cache consistency has an implementation in which both operations are wait-free.

We now show that the upper bound for sequential consistency is tight, by demonstrating the existence of two implementations. The first has a $\left\lfloor \frac{N-1}{2} \right\rfloor$-resilient write operation and a wait-free read operation. The second has a wait-free write operation and a $\left\lfloor \frac{N-1}{2} \right\rfloor$-resilient read operation. The implementations each use variables of type **Register** for the non-wait-free operation. These are instances of the linearizable registers of Section 3.4.1. Each register holds a value of type **Word**. Both implementations have a function that is identified as "periodic." These functions are needed for liveness only. Omitting them yields an implementation that is correct for finite histories.

Theorem 23. *There exists an implementation of sequential consistency with a* $\left\lfloor \frac{N-1}{2} \right\rfloor$*-resilient write operation and wait-free read and snapshot operations.*

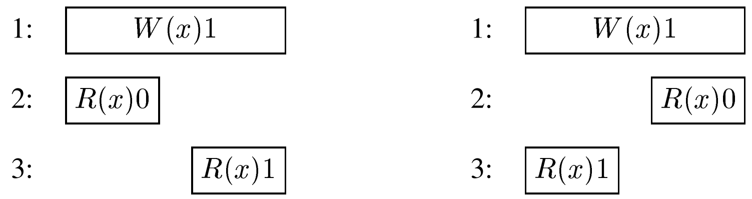
Proof. Consider the implementation of Figure 4. The lines marked as atomic in `Write` and `Snapshot` update the *LocalMem* variable in a single atomic step, once the snapshot of the underlying atomic registers has completed. Hence, the atomic action is purely local, so it will complete in finite time on any correct process. The read and snapshot operations simply return local information, so they are clearly wait-free.

The write operation may be forced to wait for the periodic code. The only nonlocal operation performed by the periodic code is a `Snapshot` of the underlying atomic registers, which is $\left\lfloor \frac{N-1}{2} \right\rfloor$-resilient by Theorem 21. Since the write operation also calls `Snapshot`, the periodic function is at least as resilient as



(a) Linearizable history (b) Non-linearizable history

Figure 1: Retimed write operations



(a) Linearizable history (b) Non-linearizable history

Figure 2: Retimed read operations

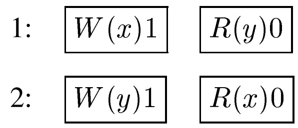


Figure 3: Non-sequential history

```
var  Mem : shared array[1 ... M] of Register;
     LocalMem : array[1 ... M] of Word;
initially
     ⟨∀x :: Mem[x] = 0 ∧ LocalMem[x] = 0⟩

function Read(x : Var) returns Word
begin
     return LocalMem[x];
end

function Snapshot() returns array[1 ... M] of Word
begin
     return LocalMem;
end

function Write(x : Var, v : Word)
begin
     Wait for any execution of Periodically to complete
     Mem[x].Write(v);
     LocalMem := Mem.Snapshot();       /* Atomic */
end

Periodically (always eventually):
begin
     Wait for any execution of Write to complete
     LocalMem := Mem.Snapshot();       /* Atomic */
end
```

Figure 4: *Fast Read* Sequential Consistency

`Write`. The main part of the write operation consists of a write to a linearizable register, which is $\left\lfloor \frac{N-1}{2} \right\rfloor$-resilient by Theorem 21, and an atomic snapshot of all linearizable registers. Therefore, the entire operation is $\left\lfloor \frac{N-1}{2} \right\rfloor$-resilient. It remains to show that the algorithm implements sequential consistency.

Due to the underlying atomic registers, the write operations alone are linearizable; that is, they appear to occur in some linear fashion. Since each write ends by updating a local copy of the shared memory via an atomic snapshot, and the periodic function updates the local copy via an atomic snapshot, the local copy is always consistent. Hence, the read operations appear to take place in serial order just after the most recent write (or run of the periodic function) by the local process. □

Theorem 24. *There exists an implementation of sequential consistency with a wait-free write operation and* $\left\lfloor \frac{N-1}{2} \right\rfloor$*-resilient read and snapshot operations.*

Proof. Consider the implementation of Figure 5. The write operation is wait-free, since it consists of one local access. The `Read` and `Snapshot` operations may block waiting for the periodic code to complete. The only non-local code executed by the periodic code is a write to a linearizable register. Since that operation is $\left\lfloor \frac{N-1}{2} \right\rfloor$-resilient, by Theorem 21, the periodic code in its entirety is $\left\lfloor \frac{N-1}{2} \right\rfloor$-resilient. Since the only non-local operations in the read operation are reads and writes of linearizable registers, the entire read operation is also $\left\lfloor \frac{N-1}{2} \right\rfloor$-resilient by Theorem 21. The snapshot operation is $\left\lfloor \frac{N-1}{2} \right\rfloor$-resilient by a similar argument. It remains to show that the algorithm implements sequential consistency.

Write operations cause an entry to be made on the $WriteQ$ queue. Queue entries are dequeued in 3 places: individually in the periodic code, and en masse during a `Read` or `Snapshot` operation. In each case, a dequeued value is immediately written into a linearizable register. Since $WriteQ$ is a queue, write operations by each individual process are seen everywhere in the order they were invoked, by the linearizability of the underlying registers. Furthermore, since reads and snapshots empty $WriteQ$ before reading, they see all previous writes by the same process. Furthermore, since reads and snapshots are snapshots of the underlying linearizable memory, they appear to take effect in some linear order. Therefore, the required linearization of operations on the underlying linearizable shared memory is a serialization of the operations, since writes may appear out of order with respect to real time. $\square$

3.4.3 Processor Consistency (PCG)

Processor consistency was first described by Goodman [18], but later researchers put varying interpretations on his definition (e.g., [2, 17, 39]). Some interpretations are equivalent to consistency conditions we treat in other sections. We consider two varieties of processor consistency, called PCG and PCD by Ahamad et al. in [2]. The authors use this terminology to distinguish the version used in the DASH shared memory computer (PCD) from the definition they suggest Goodman originally intended (PCG). We first consider PCG.

In PCG, the order of operations of each individual process is preserved, and all processes see operations on the same variable in the same order (coherence). Formally, let H_{i+w} represent $ops\ (H_i)$ together with all the write operations of H. For each process i, there is a serialization S_i of H_{i+w} such that:

- $\left\langle \forall op_1, op_2 \in H_{i+w} :: \left\langle \exists j :: op_1 \xrightarrow{j} op_2 \right\rangle \Rightarrow op_1 \xrightarrow{S_i} op_2 \right\rangle$;
- for each shared variable x, all write operations to x appear in the same order in every serialization. We write this as $S_i|_x = S_j|_x$, where $H|_x$ is the subhistory of H consisting of operations on x.

We now show that PCG has the same fault-tolerance bounds as sequential consistency in an asynchronous system.

Theorem 25. *In a DSM implementation guaranteeing processor consistency (PCG) with at least three shared variables, either the read operation or the write operation is at most* $\left\lfloor \frac{N-1}{2} \right\rfloor$*-resilient.*

Proof. Consider the history of Figure 6, the merge of two solo executions. We now show that this history is not PCG. The result then follows from Theorem 15.

For the history to be PCG, there must be serializations for each of the two processes such that they agree on the order of writes to each variable, and program order is respected. There are two cases:

- $W_1(y)2 \rightarrow W_2(y)3$. Then, to respect program order, $W_1(x)1 \rightarrow W_2(y)3$. But then the final read of x by 2 must return 1, not zero.

- $W_2(y)3 \rightarrow W_1(y)2$. Then, to respect program order, $W_2(z)1 \rightarrow W_1(y)2$. But then the final read of z by 1 must return 1, not zero.

□

Like sequential consistency, PCG collapses to cache consistency when only one variable is implemented, so it has a completely wait-free implementation in that case. It is possible to show that PCG also has a wait-free implementation when only two variables are implemented. However, this result is of little significance, so we omit it.

PCG is a strictly weaker condition than sequential consistency [2]. Thus, any implementation of sequential consistency is also an implementation of PCG.

Theorem 26. *There exist implementations of processor consistency (PCG) with:*

- *a $\left\lfloor \frac{N-1}{2} \right\rfloor$-resilient write operation and wait-free read and snapshot operations; and*

- *a wait-free write operation and $\left\lfloor \frac{N-1}{2} \right\rfloor$-resilient read and snapshot operations.*

Proof. The implementations of Theorems 23 and 24 are sequentially consistent, and thus PCG, and meet the stated resilience bounds. □

3.4.4 Processor Consistency (PCD)

The definition of PCD was intended to allow some common optimizations in shared memory machines. For example, a read operation is allowed to complete before a previous write operation on a different variable. This condition is shown in [2] to be incomparable with PCG; that is, each condition allows histories that are inconsistent for the other condition. Before we give the formal definition of PCD, we need to define the following partial orders on operations in a history:

```
type
    WriteVal = record x : Var; v : Word end;
var  Mem : shared array[1 ... M] of Register;
    WriteQ : queue of WriteVal;
initially
    ⟨∀x :: Mem[x] = 0 ∧ WriteQ[x] = ∅⟩

function Write(x : Var, v : Word)
begin
    WriteQ.enqueue (⟨x, v⟩) ;
end

function Read(x : Var) returns Word
begin
    return Snapshot()[x];
end
```

```
function Snapshot() returns array[1 ... M] of Word
begin
    Wait for any execution of Periodically to complete
    while (¬ WriteQ.empty()) do
        ⟨y, v⟩ := WriteQ.dequeue();
        Mem[y].Write(v);
    od;
    return Mem.Snapshot();
end

Periodically (always eventually):
begin
    Wait for any execution of Read or Snapshot
        to complete
    if (¬ WriteQ.empty()) then
        ⟨y, v⟩ := WriteQ.dequeue();
        Mem[y].Write(v);
    fi;
end
```

Figure 5: *Fast Write* Sequential Consistency

$$n_1 : \quad \boxed{W(x)1} \quad \boxed{W(y)2} \quad \boxed{R(z)0}$$

$$n_2 : \quad \boxed{W(z)1} \quad \boxed{W(y)3} \quad \boxed{R(x)0}$$

Figure 6: Non-PCG history

- The *partial program order* $\xrightarrow{ppo}$: If $\left\langle \exists i :: o_1 \xrightarrow{i} o_2 \right\rangle$, that is, o_1 precedes o_2 in program order, then $o_1 \xrightarrow{ppo} o_2$, unless o_1 is a write of some variable x and o_2 is a read of some distinct variable y.

- The *weak writes-before order* $\xrightarrow{wwb}$: If o_1 is a write of x, o_2 is a write of y, o_3 reads the value of y written by o_2, and $o_1 \xrightarrow{ppo} o_2$, then $o_1 \xrightarrow{wwb} o_3$.

- The *weak reads-before order* $\xrightarrow{wrb}$: This order is only defined when there is a coherence order; that is, all operations on variable x are seen in the same order S_x everywhere. If o_1 is a read of x, o_2 is a write of x such that $o_1 \xrightarrow{S_x} o_2$, o_3 is a write of y, and $o_2 \xrightarrow{ppo} o_3$, then $o_1 \xrightarrow{wrb} o_3$.

- The *semi-causal order* $\xrightarrow{sem}$: This order is the transitive closure of the union of $\xrightarrow{ppo}$, $\xrightarrow{wwb}$, and $\xrightarrow{wrb}$.

In PCD, the semi-causal order is preserved, and all processes see operations on the same variable in the same order (coherence). Formally, a history H is PCD if it satisfies the following:

- For each shared variable x, there exists a serialization S_x of $H|_x$, the subhistory of H consisting of operations on x, such that, if op_1 and op_2 are two operations in $H|_x$ and $\left\langle \exists i :: op_1 \xrightarrow{i} op_2 \right\rangle$, then $op_1 \xrightarrow{x} op_2$ (i.e., H is coherent).

- For each process i, there is a serialization S_i of H_{i+w} such that:
  - $\left\langle \forall op_1, op_2 \in H_{i+w} :: op_1 \xrightarrow{sem} op_2 \Rightarrow op_1 \xrightarrow{S_i} op_2 \right\rangle$; and
  - For any two operations op_1 and op_2 on the same variable x, $op_1 \xrightarrow{sem} op_2$ iff $op_1 \xrightarrow{x} op_2$.

We now show that PCD has the same fault-tolerance bounds as both PCG and sequential consistency in an asynchronous system.

Theorem 27. *In a DSM implementation guaranteeing processor consistency (PCD) with at least three shared variables, either the read operation or the write operation is at most $\left\lfloor \frac{N-1}{2} \right\rfloor$-resilient.*

Proof. Consider the history of Figure 7, the merge of two solo executions. We now show that this history is not PCD. The result then follows from Theorem 15.

For the history to be PCD, there must be serializations S_1, S_2, S_x, S_y, and S_z with the properties listed above. First note that $\xrightarrow{i}$ totally orders the operations on each process, since both of them follow the last write with a read of the same variable. Thus, $\xrightarrow{sem}$ also totally orders the operations on each process. Now, $\xrightarrow{wrb}$ orders $R_1(z)0$ before $W_2(y)1$; likewise, $R_2(x)0$ is ordered by $\xrightarrow{wrb}$ before $W_1(y)1$. Hence, there is a cycle in $\xrightarrow{sem}$, so PCD serializations for S_1 and S_2 do not exist. □

Like sequential consistency and PCG, PCD collapses to cache consistency when only one variable is implemented, so it has a completely wait-free implementation in that case. Whether the result of Theorem 27 holds for PCD with two variables is an open question.

PCD is a strictly weaker condition than sequential consistency [2]. Thus, any implementation of sequential consistency is also an implementation of PCD.

Theorem 28. *There exist implementations of processor consistency (PCD) with:*

- *a $\left\lfloor \frac{N-1}{2} \right\rfloor$-resilient write operation and wait-free read and snapshot operations; and*
- *a wait-free write operation and $\left\lfloor \frac{N-1}{2} \right\rfloor$-resilient read and snapshot operations.*

Proof. The implementations of Theorems 23 and 24 are sequentially consistent, and thus PCD, and meet the stated resilience bounds. □

3.4.5 Release Consistency

Release consistency was defined as part of the development of the Stanford DASH architecture [17][2]. The basic idea is that synchronization accesses are differentiated from ordinary data accesses. The former conflict with each other; the latter frequently do not. Shared memory accesses are labeled as *ordinary* or *special*. Special accesses are further divided into *synchronization* and *nonsynchronization* accesses. Finally, the synchronization accesses are divided into *acquire* (read) and *release* (write) accesses.

[2]However, DASH implemented something weaker than release consistency; it did not always guarantee coherence [2].

Ordinary accesses are bracketed between acquire-release pairs. That is, an acquire is made visible everywhere before any following ordinary accesses. Ordinary accesses are made visible everywhere before any following releases. Release consistency comes in two varieties. In RC_{SC}, the special accesses are sequentially consistent with respect to one another. In RC_{PC}, the special accesses are processor consistent (PCD) with respect to one another. Finally, there is a coherence requirement. Accesses to any individual shared variable x are seen in the same order everywhere.

Release consistency has been implemented in several settings. Some implementations (such as Munin [10], TreadMarks [25], and CASHMERe [27]) actually implement critical section, or locking, semantics. That is, an acquire corresponds to a lock operation, and a release to an unlock operation. This is strictly stronger than the formal definition. Furthermore, in a totally asynchronous setting, such an implementation must have 0-resilient synchronization accesses, since a process may fail while in a critical section. Since failed processes and slow processes cannot be distinguished, such an untimely failure blocks all other processes from completing acquire operations.

When all operations are synchronization accesses, release consistency reduces to either sequential consistency or PCD. Since both conditions have the same upper bound, we immediately get an upper bound for release consistency.

Theorem 29. *In a DSM implementation guaranteeing release consistency with at least two (RC_{SC}) or three (RC_{PC}) shared variables, either the acquire or the release operation is at most $\left\lfloor \frac{N-1}{2} \right\rfloor$-resilient.*

Proof. The result follows immediately from Theorems 22 and 27 when all operations are acquires and releases. □

We now show that this bound is tight by demonstrating an implementation that achieves it for RC_{SC}. Since PCD is strictly weaker than sequential consistency, this implementation also implements RC_{PC}. In this implementation, ordinary accesses are wait-free; one of acquire and release is wait-free, and the other is $\left\lfloor \frac{N-1}{2} \right\rfloor$-resilient. We omit an implementation of `Snapshot`, since it is trivial in the ordinary case, and a one-line difference from acquire in the synchronized case.

$$
\begin{array}{lcccc}
1: & \boxed{W(x)1} & \boxed{W(y)1} & \boxed{R(y)1} & \boxed{R(z)0} \\
2: & \boxed{W(z)1} & \boxed{W(y)1} & \boxed{R(y)1} & \boxed{R(x)0}
\end{array}
$$

Figure 7: Non-PCD history

```
type
    WriteVal = record x : Var; v : Word end;
var
    Mem : shared SeqConsMem[1 ... M];
    LocalMem : array[1 ... M] of Word;
    WriteQ : queue of WriteVal;
initially
    (∀x :: Mem[x] = 0 ∧ LocalMem[x] = 0) ∧
    WriteQ = ∅

function Read(x : Var) returns Word
begin
    return LocalMem[x];
end

function Acquire(x : Var) returns Word
begin
    WriteQ := ∅;
    LocalMem := Mem.Snapshot();
    return LocalMem[x];
end
```

```
function Write(x : Var, v : Word)
begin
    LocalMem[x] := v;
    WriteQ.enqueue (⟨x, v⟩) ;
end

function Release(x : Var, v : Word)
begin
    Wait for any execution of Periodically
        to complete
    while (¬ WriteQ.empty()) do
        ⟨y, w⟩ := WriteQ.dequeue();
        Mem[y].Write(w);
    od;
    Mem[x].Write(v);
end

Periodically (always eventually):
begin
    Wait for any execution of Release to complete
    if (¬ WriteQ.empty()) then
        ⟨y, v⟩ := WriteQ.dequeue();
        Mem[y].Write(v);
    fi;
end
```

Figure 8: Release Consistency implementation for process i

Theorem 30. *There exist implementations of release consistency with one $\left\lfloor \frac{N-1}{2} \right\rfloor$-resilient synchronization operation, and all other operations wait-free.*

Proof. Consider the implementation of Figure 8. It uses an implementation of sequential consistency (type $\mathbf{SeqConsMem}$). Note that the read and write operations consist of purely local operations; hence, they are wait-free. The acquire operation has only one non-local operation, a call to `Snapshot` on the sequentially consistent memory. Hence, its resilience is equal to the resilience of the snapshot operation for the sequential consistency implementation. The release operation executes a bounded number of write operations on the sequential consistency implementation; hence its resilience is equal to that of the sequential consistency write operation. Therefore, by using the two sequential consistency implementations of Theorems 23 and 24, we achieve the stated resilience bounds. Our remaining proof obligation is to show that release consistency is implemented correctly.

Since the synchronization accesses use the sequential consistency implementation, they are sequentially consistent with respect to one another. Ordinary accesses made outside of an acquire/release pair are not guaranteed to be made visible anywhere; hence, the emptying of the write queue in the acquire code is correct. Since ordinary write operations are stored in the write queue, any writes made before a release are made visible elsewhere during the release, due to being written into the sequentially consistent memory. Ordinary read operations read from a consistent snapshot of the memory; hence, they are correct. Finally, writes made "before" an acquire (in the sequential consistency ordering) are made visible at the time of the acquire due to the snapshot. $\square$

3.4.6 Hybrid Consistency

Hybrid consistency [7] is similar to release consistency, in that there are two kinds of operations. In this case, they are called *strong* and *weak* operations. The strong operations are either linearizable or sequentially consistent with respect to one another. The weak operations are ordered only by the strong operations they lie between. That is, for HC_{LIN} (hybrid consistency based on linearizability), for each process i, there is a serialization S_i of H_{i+w} such that:

- for all strong operations sop_1, sop_2, all processes see the operations in the same order; that is, $\left\langle \forall i :: sop_1 \xrightarrow{S_i} sop_2 \right\rangle \vee \left\langle \forall i :: sop_2 \xrightarrow{S_i} sop_1 \right\rangle$;

- for all operations op^i_1, op^i_2 invoked by process i such that $op^i_1 \xrightarrow{i} op^i_2$, if one of the two operations is strong, then $\left\langle \forall j :: op^i_1 \xrightarrow{S_j} op^i_2 \right\rangle$; and

- there exists a linearization L_s of all strong operations such that for any two strong operations sop_1, sop_2, if $sop_1 \xrightarrow{L_s} sop_2$, then $\left\langle \forall i :: sop_1 \xrightarrow{S_i} sop_2 \right\rangle$.

The condition for HC_{SC} (hybrid consistency based on sequential consistency) is nearly identical, except that a serialization is required in the third condition, rather than a linearization.

Published implementations of hybrid consistency [7, 16] use techniques that are 0-resilient for asynchronous systems, such as atomic broadcast. In this section, we show that hybrid consistency can be implemented in a fault-tolerant manner on such systems.

When all operations are strong, hybrid consistency reduces to either sequential consistency or linearizability. This immediately gives us a pair of upper bounds for hybrid consistency.

Theorem 31. *In a DSM implementation guaranteeing HC_{LIN} on at least 3 processes, the strong operations are both at most $\left\lfloor \frac{N-1}{2} \right\rfloor$-resilient. In a DSM implementation guaranteeing HC_{SC} with at least two shared variables, one of the strong operations is at most $\left\lfloor \frac{N-1}{2} \right\rfloor$-resilient.*

Proof. The result follows immediately from Theorems 19, 20, and 22 when all operations are strong. ☐

We now show that this bound is tight by demonstrating an implementation. In each, weak accesses are wait-free, and the strong operations depend on the underlying implementation of sequential consistency or linearizability. The lower bound then follows directly from Theorems 21, 23, and 24. As with release consistency, we omit the definition of the snapshot operation, since it is a trivial alteration to the two read operations.

```
type
    WriteVal = record x : Var; v : Word end;
var  Mem : shared LINorSCMem[1 ... M];
     LocalMem : array[1 ... M] of Word;
     WriteSet : set of WriteVal;
initially
    ⟨∀x :: Mem[x] = 0 ∧ LocalMem[x] = 0⟩ ∧
    WriteSet = ∅

function WeakRead(x : Var) returns Word
begin
    return LocalMem[x];
end

function StrongRead(x : Var) returns Word
begin
    LocalMem := Snapshot(Mem);
    for ⟨y, w⟩ ∈ WriteSet do
        LocalMem[y] := w;
    od;
    return LocalMem[x];
end

function WeakWrite(x : Var, v : Word)
begin
    LocalMem[x] := v;
    WriteSet := WriteSet ∪ {⟨x, v⟩} ;
end

function StrongWrite(x : Var, v : Word)
begin
    while ⟨∃ ⟨y, w⟩ ∈ WriteSet⟩ do
        WriteSet := WriteSet − ⟨y, w⟩
        Mem[y].Write(w);
    od;
    Mem[x].Write(v);
end

Periodically (always eventually):
begin
    if ⟨∃ ⟨y, w⟩ ∈ WriteSet⟩ then
        Mem[y].Write(w);
    fi;
    LocalMem := Snapshot(Mem);
end
```

Figure 9: Hybrid Consistency implementation for process i

Theorem 32. *There exist implementations of HC_{LIN} with $\left\lfloor \frac{N-1}{2} \right\rfloor$-resilient strong operations and wait-free weak operations. There exist implementations of HC_{SC} with one strong operation $\left\lfloor \frac{N-1}{2} \right\rfloor$-resilient and all other operations wait-free.*

Proof. Consider the implementation of Figure 9. It assumes the availability of an implementation of either linearizability or sequential consistency (type $\mathbf{LINorSCMem}$). The weak read and write operations consist of purely local operations; hence, they are wait-free. The strong read operation has only one non-local operation, a call to `Snapshot` on the underlying linearizable or sequentially consistent memory. Hence, its resilience is equal to the resilience of the snapshot operation for that implementation. The strong write operation executes a bounded number of write operations on the sequentially consistent (linearizable) memory; hence its resilience is equal to that of the sequential consistency (linearizability) write operation. Therefore, by using the sequential consistency implementations of Theorems 23 and 24, and the linearizability implementation of Theorem 21, we achieve the stated resilience bounds. Our remaining proof obligation is to show that hybrid consistency is correctly implemented.

Since the strong operations use the sequential consistency (linearizability) implementation, they are sequentially consistent (linearizable) with respect to one another. Since weak writes are stored in the write queue, any writes made between strong operations are made visible elsewhere by the beginning of the next strong write operation. Weak reads return values from a consistent snapshot of the memory; hence, they are correct. □

3.4.7 Wait-free conditions

In this section we show that several weaker memory models have implementations in which all operations are wait-free. In particular, we show that causal memory, pipelined RAM, mixed consistency, and cache consistency all have completely wait-free implementations.

Causal Memory Causal memory [3] is a weak memory that preserves the *causal order*. Whenever a write operation W becomes visible locally, all write operations that may have influenced it (i.e., all write operations whose effects may have been seen by the originating process of W) must be visible locally as well. The implementation given by Ahamad et al. in [3] is based on unbounded timestamps. Both the read and write operations are wait-free. The implementation continues to provide safety in the presence of faults, but may fail to make progress if a process fails in the middle of a broadcast. Those processes that receive the broadcast may eventually accept the write, placing it in their causal past. Any further writes by those processes cannot be accepted by the others, since they cannot see some write in the causal past. An alternative is to simply transmit the entire causal past with each write. While allowing progress in the presence of faults, this approach produces messages of unbounded size. Garbage collection can be applied to remove writes that are known to be visible everywhere. However, when a process fails, the other processes' messages will grow without bound, unable to garbage collect. Whether there exists a bounded implementation that makes progress in the presence of faults is an open question.

Pipelined RAM Pipelined RAM (PRAM) [33] is a strictly weaker condition than causal memory. It requires that any two writes issued from the same processor are seen everywhere in the order they were invoked. Since causal memory has a wait-free implementation, so does PRAM. However, the implementation given by Lipton and Sandberg [33] is bounded, and continues to make progress in the presence of faults. Each process simply broadcasts its writes over FIFO channels as they are invoked, and applies writes to its local copy of the shared memory as they arrive.

Mixed Consistency Mixed consistency [1] is a combination of causal memory and PRAM. Reads are either labeled as causal, in which case they see the entire causal past, or PRAM, in which case they see all program orders. Since mixed consistency reduces to causal memory when all reads are labeled as causal, mixed consistency also has an implementation in which both operations are wait-free.

Cache Consistency (coherence) Cache consistency or coherence [18] requires that there be a global order on the writes to any particular shared variable x. However, two processes can observe different interleavings of writes to different variables. This condition has a simple wait-free implementation based on unbounded timestamps. It is shown in Figure 10.

The `next` macro produces monotonically increasing values on each process, such that values produced on distinct processes are disjoint. These values are used to impose a total ordering on the writes to any one shared location. A $\mathbf{MemVal}$ holds a value for a variable and an associated timestamp. A $\mathbf{Message}$ holds a shared variable name and a $\mathbf{MemVal}$ to store there. The variable *Mem* holds a local copy of the shared space.

On a write of v to x, the local value of x is updated, and the local timestamp on x is increased. The new value of x (and the new timestamp) are then broadcast to all other processes by function `BCast`, which can be a true broadcast or multiple point-to-point sends. On a read of x, the local value of x is returned. On receiving a message, function `MessageHandler` checks whether the timestamp on the message is higher than the stored timestamp for that variable. If so, the new value is stored; otherwise, it is discarded, as it has been (conceptually) "overwritten" by the write with the higher timestamp..

This implementation is clearly wait-free. To show that it implements cache consistency, we simply note that there is a total ordering on the timestamps associated with any one variable, and that each process obeys that timestamp order. That is, a value is only overwritten by one with a higher timestamp. Thus the timestamp order is the coherence order.

4 Related Work

Mavronicolas and Roth [38] give two implementations of linearizable shared memory for a partially synchronous system. In such a system, all message latencies are in the range $[d - u, d]$ for known constants d (delay) and u (uncertainty), $0 \leq u < d$. Like us, they consider read and write operations separately, so that one operation can be optimized in cases where it is used more frequently. They derive lower bounds on the response time of sequentially consistent and linearizable DSMs in terms of d and u. Our results are for an asynchronous system, in which d is unbounded.

Attiya and Welch [8] compared linearizability and sequential consistency, and found that sequential consistency can be implemented more efficiently in a partially synchronous setting. In particular, they showed that either the read or the write operation of sequential consistency must suffer at least one network latency, and both operations of linearizability must do so. In the synchronous case, both linearizability and sequential consistency can have one instantaneous operation and one that suffers a network latency.

Kosa [28] compares the costs of linearizability, sequential consistency, and hybrid consistency of ADTs in terms of worst-case time complexity. In addition to the consistency condition, two factors were considered: the synchrony of the system (synchronous or partially synchronous); and the algebraic properties of a given ADT. As in our analysis, Kosa found that a tradeoff occurs; improving one operation may require making other operations worse. The author also showed that hybrid consistency does not necessarily perform better than sequential consistency for arbitrary types.

Friedman [16] gave implementations of hybrid consistency with synchronization operations, in addition to the usual read and write operations. He derived lower bounds on the time required to implement several synchronization operations in a partially asynchronous setting when $d = 0$ (i.e., message delays are in the range $[0, u]$). The bounds apply to weak consistency, so the lower bounds apply to all strictly stronger conditions, such as hybrid consistency. The implementations given in [16] rely on atomic broadcast, which we cannot use since it has only 0-resilient implementations in asynchronous systems [15].

Malki et al. [36] study the *Dynamic Uniformity* problem. In this problem, a set of actions A is identified. Whenever any node in a particular group takes an action in A, all other nodes in the group must eventually take that same action, even if the first node subsequently fails. Moreover, nodes can leave and join the group during the computation. The authors show that D-Uniformity is solvable if a majority of the processes do not crash; that is, they give an $\left\lfloor \frac{N-1}{2} \right\rfloor$-resilient solution to the problem. They also show that if a serial ordering constraint is added, any solution must be 0-resilient. Some of the problems we consider are similar, but there is no dynamic group membership. We show that the sequential consistency and linearizability problems, which impose a serial ordering constraint but have static membership, have $\left\lfloor \frac{N-1}{2} \right\rfloor$-resilient solutions.

5 Conclusion

We have derived general conditions on consistent and inconsistent histories that together imply fault tolerance bounds for oblivious consistency conditions in asynchronous systems. However, these results also apply to other objects in the partially asynchronous model when the upper bound on message latencies is unknown. Furthermore, the upper bound proofs do not depend on the semantics of read, write, and snapshot operations. They can be applied equally well to other kinds of operations. Therefore, the upper bound theorems can be used to derive fault tolerance bounds for other kinds of objects besides DSMs.

We showed that the upper bounds are tight for many previously studied consistency conditions. We showed that any implementation of linearizability has read and write operations that are at most $\left\lfloor \frac{N-1}{2} \right\rfloor$-resilient, and that an implementation exists that achieves this bound. We showed that any implementation of sequential consistency or either kind of processor consistency has at least one operation that is at most $\left\lfloor \frac{N-1}{2} \right\rfloor$-resilient, and that this is achievable. We also showed that release (hybrid) consistency must have one synchronization (strong) operation that is at most $\left\lfloor \frac{N-1}{2} \right\rfloor$-resilient, and that these bounds are achievable. Finally, we showed that several kinds of weak shared memory have entirely wait-free implementations.

The proof technique we used to show the general upper bounds is of independent interest. We have since used the same technique to investigate the existence of other consistency conditions, besides linearizability and cache consistency, that are *local properties*. Such consistency conditions are of interest, since their implementations do not require any cross-variable synchronization. We will continue investigating other uses of this proof technique, especially whether it can be used to show resource (time, space) bounds on DSM implementations. We will also investigate extensions of such results to more general objects.

References

[1] D. Agrawal, M. Choy, H. V. Leong, and A. K. Singh. Mixed consistency: A model for parallel programming. In *PODC '94*, pages 101–10, Los Angeles, CA, USA, 14–17 Aug. 1994.

[2] M. Ahamad, R. Bazzi, R. John, P. Kohli, and G. Neiger. The power of processor consistency. In *SPAA '93*, pages 251–60, Velen, Germany, 30 June–2 July 1993.

[3] M. Ahamad, G. Neiger, J. E. Burns, P. Kohli, and P. W. Hutto. Causal memory: Definitions, implementation and programming. *Distrib. Comput.*, 9(1):37–49, Aug. 1995.

[4] H. Attiya. Efficient and robust sharing of memory in message-passing systems. In Ö. Babaoglu and K. Marzullo, editors, *WDAG '96*, volume 1151 of *Lecture Notes in Computer Science*, pages 56–70, Bologna, Italy, 9–11 Oct. 1996. Springer-Verlag.

[5] H. Attiya, A. Bar-Noy, D. Dolev, D. Peleg, and R. Reischuk. Renaming in an asynchronous environment. *J. ACM*, 37(3):524–48, July 1990.

[6] H. Attiya and R. Friedman. Limitations of fast consistency conditions for distributed shared memories. *Inf. Process. Lett.*, 57(5):243–8, Mar. 1996.

[7] H. Attiya and R. Friedman. A correctness condition for high-performance multiprocessors. *SIAM J. Comput.*, 27(2), Apr. 1998.

[8] H. Attiya and J. L. Welch. Sequential consistency versus linearizability. *ACM Trans. Comput. Syst.*, 12(2):91–122, May 1994.

[9] E. Borowsky and E. Gafni. Generalized FLP impossibility result for t-resilient asynchronous computations (extended abstract). In *STOC '93*, pages 91–100, San Diego, CA, USA, 16–18 May 1993.

[10] J. B. Carter. Design of the Munin distributed shared memory system. *J. Parallel Distrib. Computing*, 29(2):219–27, Sept. 1995.

[11] T. Chandra, V. Hadzilacos, P. Jayanti, and S. Toueg. Wait-freedom vs. t-resiliency and the robustness of wait-free hierarchies (extended abstract). In *PODC '94*, pages 334–43, Los Angeles, CA, USA, 14–17 Aug. 1994.

[12] T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *J. ACM*, 43(2):225–67, Mar. 1996.

[13] F. Cristian. Reaching agreement on processor group membership in synchronous distributed systems. *Distrib. Comput.*, 4(4):175–87, Apr. 1991.

[14] D. Dolev, C. Dwork, and L. Stockmeyer. On the minimal synchronism needed for distributed consensus. *J. ACM*, 34(1):77–97, Jan. 1987.

[15] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–82, Apr. 1985.

[16] R. Friedman. Implementing hybrid consistency with high-level synchronization operations. *Distrib. Comput.*, 9(3):119–29, Dec. 1995.

[17] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *ISCA '90*, pages 15–26, Seattle, WA, USA, 28–31 May 1990. Revised in Stanford CSL Technical Report CSL-TR-93-568.

[18] J. R. Goodman. Cache consistency and sequential consistency. Technical Report 1006, Computer Sciences Department, University of Wisconsin-Madison, Feb. 1991.

[19] M. P. Herlihy. Wait-free synchronization. *ACM Trans. Prog. Lang. Syst.*, 13(1):124–49, Jan. 1991.

[20] M. P. Herlihy and N. Shavit. The asynchronous computability theorem for t-resilient tasks. In *STOC '93*, pages 111–20, San Diego, CA, USA, 16–18 May 1993.

[21] M. P. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Prog. Lang. Syst.*, 12(3):463–92, July 1990.

[22] M. Inoue, W. Chen, T. Masuzawa, and N. Tokura. Linear-time snapshot using multi-writer multi-reader registers. In G. Tel and P. Vitányi, editors, *WDAG '94*, volume 857 of *Lecture Notes in Computer Science*, pages 130–40, Terschelling, The Netherlands, 29 Sept.–1 Oct. 1994. Springer-Verlag.

[23] A. Israeli, A. Shaham, and A. Shirazi. Linear-time snapshot implementations in unbalanced systems. *Math. Systems Theory*, 28(5):469–86, Sept.–Oct. 1995.

[24] J. James and A. K. Singh. Fault tolerance bounds for memory consistency. In M. Mavronicolas and P. Tsigas, editors, *WDAG '97*, volume 1320 of *Lecture Notes in Computer Science*, pages 200–14, Saarbrücken, Germany, 24–26 Sept. 1997.

[25] P. Keleher, A. L. Cox, S. Dwarkadas, and W. Zwaenepoel. An evaluation of software-based release consistent protocols. *J. Parallel Distrib. Computing*, 29(2):126–141, Oct. 1995.

[26] P. Kohli, G. Neiger, and M. Ahamad. A characterization of scalable shared memories. In C. Y. R. Chen and P. B. Berra, editors, *ICPP '93*, volume 1: Architecture, pages 332–5, Saint Charles, IL, USA, 16–20 Aug. 1993.

[27] L. I. Kontothanassis, M. L. Scott, and R. Bianchini. Lazy release consistency for hardware-coherent multiprocessors. In *Supercomputing '95*, pages 1705–36, San Diego, CA, USA, 3–8 Dec. 1995.

[28] M. J. Kosa. Making operations of concurrent data types fast. In *PODC '94*, pages 32–41, Los Angeles, CA, USA, 14–17 Aug. 1994.

[29] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Comput.*, 28(9):690–1, Sept. 1979.

[30] L. Lamport. On interprocess communication, parts I and II. *Distrib. Comput.*, 1(2):77–101, Apr. 1986.

[31] K. Li and P. Hudak. Memory coherence in shared virtual memory systems. *ACM Trans. Comput. Syst.*, 7(4):321–59, Nov. 1989.

[32] M. Li, J. Tromp, and P. M. B. Vitányi. How to share concurrent wait-free variables. *J. ACM*, 43(4):723–46, July 1996.

[33] R. J. Lipton and J. S. Sandberg. PRAM: A scalable shared memory. Technical Report CS-TR-180-88, Department of Computer Science, Princeton University, Sept. 1988.

[34] W.-K. Lo. More on t-resilience vs. wait-freedom (extended abstract). In *PODC '95*, pages 110–19, Ottawa, Ontario, Canada, 20–23 Aug. 1995.

[35] N. A. Lynch. *Distributed Algorithms*. Morgan Kaufmann Series in Data Management Systems. Morgan Kaufmann Publishers, San Francisco, CA, 1996.

[36] D. Malki, K. Birman, A. Ricciardi, and A. Schiper. Uniform actions in asynchronous distributed systems. In *PODC '94*, pages 274–83, Los Angeles, CA, USA, 14–17 Aug. 1994.

[37] M. Mavronicolas and D. Roth. Sequential consistency and linearizability: Read/write objects. In *Proc. 29th Annual Allerton Conf. on Communication, Control, and Computing*, pages 683–92, Allerton House, Monticello, IL, USA, 2–4 Oct. 1991.

[38] M. Mavronicolas and D. Roth. Efficient, strongly consistent implementations of shared memory (extended abstract). In A. Segall and S. Zaks, editors, *WDAG '92*, volume 647 of *Lecture Notes in Computer Science*, pages 346–61, Haifa, Israel, 2–4 Nov. 1992. Springer-Verlag.

[39] D. Mosberger. Memory consistency models. *Operating Systems Review*, 27(1):18–26, Jan. 1993.

[40] E. Schenk. The consensus hierarchy is not robust. In *PODC '97*, page 279, Santa Barbara, CA, USA, 21–24 Aug. 1997.

[41] A. K. Singh. A framework for programming with non-atomic memories. *J. Parallel Distrib. Computing*, 26(2):211–224, Apr. 1995.

Biographies

Jerry James is an Assistant Professor in the EECS Department of the University of Kansas. He received his M.S. in Computer Science in 1993 from Brigham Young University, and his Ph.D. in Computer Science in 1999 from the University of California, Santa Barbara. His research interests are concurrent objects, compositionality, shared data consistency, and fault-tolerant distributed systems. He is a member of the ACM, and the IEEE Computer Society.

Raimondas Lencevicius is a Senior Research Engineer at Nokia Research Center. He received his Diploma in Applied Mathematics from Vilnius University, Lithuania in 1992, and his Ph.D. in Computer Science from the University of California, Santa Barbara in 1999. His research interests are debugging and visualization of object-oriented systems, as well as run-time program analysis and transformation. He is a member of the ACM.

Ambuj Singh is an Associate Professor of Computer Science at the University of California, Santa Barbara. He received an M.S. in Computer Science from Iowa State University in 1984, and a Ph.D. in Computer Science from the University of Texas at Austin in 1989. His research interests are distributed algorithms, digital libraries, databases, and consistency, compositionality and fault-tolerance of distributed objects. He is a member of the ACM and the IEEE Computer Society.

```
macro     next(x) = N · ⌈(x+1−i)/N⌉ + i
type      MemVal = record v : Word; t : Integer end;
          Message = record x : Var; mv : MemVal end;
var       Mem : array[1..M] of MemVal
initially ⟨∀j :: Mem[j] = ⟨0,0⟩⟩

function Write(x : Integer, v : Word)
begin
    Mem[x] := ⟨v, next(Mem[x].t)⟩ ;
    BCast(x, Mem[x])
end

function Read(x : Integer) returns Word
begin
    return Mem[x].v
end

function MessageHandler(m : Message)
begin
    if Mem[m.x].t < m.mv.t then /* Atomic */
        Mem[m.x] := m.mv
    fi
end
```

Figure 10: Cache consistency implementation for process i